May 1984.

An Expandable Object-based UNIX Kernel

Erik Reeh Nielsen
Søren Lauesen
Vilhelm Rosenqvist

NCR Systems Engineering Copenhagen
Haslegade 16
DK-2100 Copenhagen
Denmark
phone: + 45 1 29 58 00
telex: 16450 ncrse dk

Abstract:

Many key features of UNIX are hidden in the UNIX-kernel. This implies that users cannot expand their system the way they need unless they modify the kernel.

A commercial UNIX* based system may require changes like the following:
- replace security policy
- add reliable database system
- add transparent remote file access
- replace or add file system to allow disc formats compatible with other operating systems.

If a source license is available, it may be feasible for one organization to make its own changes to the UNIX-kernel and to maintain the new kernel version. However, changes/extensions made by different organizations cannot easily be integrated under present kernel architectures.

This presentation shows a UNIX-kernel implementation, called the EOS Software Factory, that overcomes these problems. The UNIX-kernel is implemented as a number of separately protected modules (objects). The modules are dynamically coupled when the system executes. Modules may even be installed and removed on-the-fly.

Actually, the EOS software factory not only support UNIX, it can as well support applications written for other operating systems if the appropriate modules are added.

The EOS Software Factory has been implemented on standard hardware (Motorola 68000) and runs with competitive performance.

* UNIX is a trade-mark of Bell Laboratories.

Disclaimer

This paper explains an experimental operating system developed by NCR. It is **not** a product offering, but serves only as an introduction to the technology involved.

Introduction

EOS is the name of an object oriented software technology for decentralized operating systems. A decentralized operating system means an operating system where OS-modules like device drivers, file systems, database systems, and job handlers can be independently developed and easily combined. With the EOS technology OS-modules are separately protected, they can be user-written, they can be installed and tested on-the-fly like ordinary user programs, and they are dynamically linked together with no need for a special link or sysgen phase. To a system builder this means that he can easily tailor his system, e.g. get some OS-modules from the hardware vendor, other OS-modules from 3rd party, and add some OS-modules himself.

The EOS technology is initially available through an operating system build on these principles. This OS, called the EOS Software Factory, contains a decentralized implementation of the UNIX kernel. The Software Factory can be used to run the wealth of UNIX applications and development tools, it can be tailored for special purpose applications or support other operating systems than UNIX, and it can be used as development system for further OS-module development. The first version of the Software Factory runs on a Motorola 68000 Exormacs.

The acronym EOS stands for Expandable Operating System. In Greek the word EOS means "dawn" or the goddess of dawn.

State-of-the-Art Operating Systems

In a state-of-the-art operating system many key features, e.g. job handling, resource allocation, file management, and security management, are part of a single central operating system component usually called the kernel, executive, or supervisor. In UNIX this central component is the UNIX kernel.

Such features, that are part of the kernel, cannot be modified or extended without stopping the entire system, since changes requires recompilation or relinking of the kernel followed by a restart with the new kernel. Although the kernel internally consists of several modules, these are not protected, so an error in one module may easily harm the reliability of other modules. For these reasons changes to kernel features is a cumbersome and risky process, in particular in commercial production environments.

Features that are implemented as application (or utility) programs can, however, normally be added or modified in a much more smooth manner. Since applications normally execute in a protected environment, they can be tested without harming other parts of the system. In fact, a major function of most state-of-the-art operating systems is to allow users to develop and execute application (or utility) programs.

Most existing operating systems allow an application to communicate with other applications. This gives the application some means of providing services to other applications, i.e. act as an operating system component. UNIX has several examples of applications (utilities) serving as operating system components: The shell and printer spooler are two such examples.

The benefits from implementing operating system functions as utility programs on state-of-the-art operating systems are numerous. As explained above, OS-modules implemented as utility programs can easily be added or modified. Any user can be allowed to develop such an OS-module, since it executes in a protected environment and, thus, cannot harm other parts of the system. Several different versions of the same service can coexist, witness the various UNIX shell versions.

Unfortunately, the communication facilities between applications in state-of-the-art operating systems are insufficient or too specialized so that many OS-facilities cannot be put into utility-programs. In UNIX, for example, new OS-facilities like a communications driver, a reliable database system (with commit and roll-back), or a transparent remote file access facility cannot be implemented as utilities, they have to be integrated into the UNIX kernel.

Purpose of EOS technology

The overall purpose of the EOS technology is to make the central part of an operating system as small as possible. Most of the OS-modules become "decentral modules", that can be developed by the user or 3rd party. The remaining central part, called the EOS kernel, is the Software Bus which lets the decentral modules interface with each other.

More precisely, EOS fulfills the following requirements:

System software can be developed/added by user:
The user can combine 3rd party OS-modules, own OS-modules, and vendor modules, and he does not need the source texts of 3rd party or vendor modules.

System software modules cannot corrupt each other:
OS-modules can only interface with other modules through the EOS-kernel, and the kernel does not allow one module to change the interior part of another module. This property facilitates testing and isolates errors. Thus, it improves system reliability.

System software modules can be installed and configured on-the-fly:
An OS-module can be installed while the system is running normally. After installation, the OS-module can be instructed to configurate itself (adapt to the peripheral devices available, the expected load, etc.). An OS-module can also be removed on-the-fly; for instance, when in error or as preparation for a new version to be installed.

System software can be developed incrementally:
A new OS-module improving the services of an existing module can be installed and transparently made available to the users. The improved services can often be implemented on top of the old services without need for access to the source text of the existing module. A file system with memory cache, for example, can be built on top of an existing file system handling the disc without cache.

Several file systems, job handlers, etc. can coexist:
An EOS-based system can support several file systems; for instance handling different disc formats. Furthermore, an application can concurrently access some files through one

file system, other files through another file system. Also, several job handlers can run concurrently; for example one supporting UNIX and another supporting CP/M or MS-DOS.

EOS Technology Principles

The EOS technology is based on the object-oriented operating system technology pioneered by Hydra [Wulf 74, 81] and CAP [Wilkes 79]. A major achievement during the EOS development has been to improve this technology, in particular in the area of abortion and clean-up, and show how a decentralized, commercial OS could be organized.

The EOS technology can be characterized by the following points:

Object oriented:

An EOS module is an object or a collection of closely related objects.
An object is a collection of data and procedures operating on that data. An object can only be accessed from outside by means of the so called entry procedures. The data of an object can not be accessed directly from outside the object, but the entry procedures may return copies of the data or pointers to part of it. These rules are enforced by a combination of hardware protection and the EOS kernel or Software Bus.

"Large" objects:

EOS uses objects mainly for enforcing protection. Usually, an OS-module consists of a single or a few reasonably large objects. A file system is one object, an executing application program is one object, etc. This means that traditional programming style can be used inside objects, and that the EOS kernel only intervenes when an object (protection) boundary is crossed during execution. The result is that acceptable performance is possible with traditional hardware.

Decentralized abortion and clean-up:

A major problem in decentralizing an operating system is abortion and clean-up. As an example consider a database system engaged in servicing an application. The database system has set aside resources for the application etc. When the application terminates, the database system must perform some clean-up (make resources free, roll transaction back, etc.). In case the application terminates in error or is aborted, it cannot itself notify the database system. Inventions made as part of the EOS development and implemented in the Software Bus can be used by any OS-module to ensure that it can clean-up when serviced modules terminate.

Another problem arises when an application is waiting for some resources within the database system (maybe waiting indefinitely because of a deadlock) and the operator wants to kill the application. The database system must then be notified so that it can cancel the waiting. The EOS development has also resulted in new inventions for solving this problem in a decentralized manner.

```
object IoSystem
  entry Assign (out file; name, .. )
  entry Rename (oldName, newName )
  entry CreateLink (oldName, .. )
  entry DeleteLink (fileName )

object File
  entry Read (buffer, pos, count )
  entry Write (buffer, pos, count )
  entry SetSize (fileSize, volume)
  entry SetMode ( .. )
```

Figure 1. EOS I/O System Family

Architecture for OS-module families:

EOS shows how a complete, commercial operating system can be split into rather independent modules. In order to allow several file systems to be interchangeable, certain rules must be followed by a module qualifying as a (standard) file system. These rules define the family of file systems. How a file is organized on disc, etc. is not prescribed in the family rules. Only a set of interface procedures (entries) to the file system is part of the family standard. These interfaces are sufficient for an application program (or an OS-module) to use any file system belonging to the family. Figure 1 shows part of the I/O-System family specification.

Similar families are prescribed for resource managers, job handlers, security managers, etc.

Initial Implementation: EOS Software Factory

The EOS technology is initially available as the so-called Software Factory, an operating system built on the EOS principles. The Software Factory contains a decentralized implementation of the UNIX kernel (in the first version compatible with UNIX version 7).

The EOS-UNIX-kernel is not built into the EOS-kernel. Rather it has just been implemented as OS-modules among other OS-modules. In fact, most of the modules comprising the decentralized UNIX-kernel are standard EOS modules useful for other purposes as well.

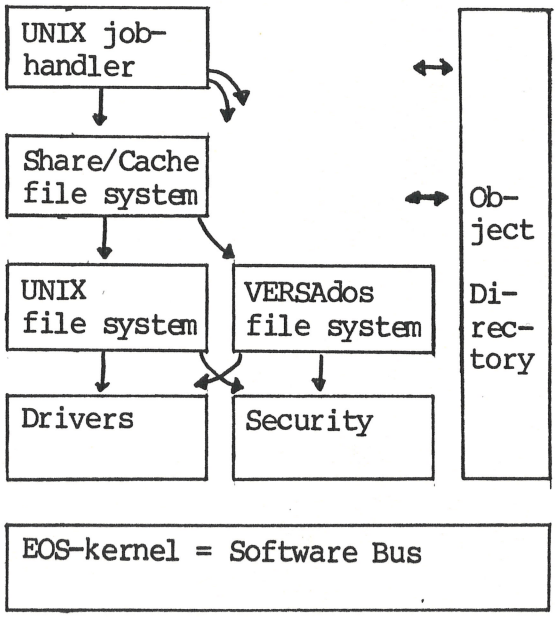

Figure 2. EOS Software Factory.

The EOS Software Factory consists of the following software modules (see figure 2):

The EOS-kernel:

This Software Bus provides the foundation which lets the decentral modules interface to each other, it enforces protection, and it includes basic memory management and process

scheduling. The basic memory management and process scheduling are used by decentral OS-modules which provide more elaborate resource management and scheduling.

Security module:
This module implements a security database for use by the other OS-modules. It is optional in the sense that meaningful defaults are assumed by the OS-modules in case the security module is not installed.

Object directory:
The object directory is a module that can load and install other modules. It also provides named cross-references between modules - subject to security restrictions in case the security module is installed.

Device drivers:
The Software Factory contains drivers for discs, terminals, and printers. Due to limitations in the Motorola 68000 EXORmacs hardware, interrupt procedures have to run in supervisor mode without memory protection. Apart from this, the driver modules are not different from other OS-modules. They can be installed and removed on-the-fly and they are written in a high-level language (EosPascal).

UNIX and VERSAdos file systems:
The software factory contains two standard low-level file systems, one supporting UNIX disc packs and another supporting Motorola VERSAdos disc packs. Both file systems can be used via normal UNIX i/o procedures.

File sharing and cache module:
UNIX-type file sharing and disc cache is implemented in a separate module, which can use any standard i/o-system for low-level file i/o. This module, the device drivers, and the file systems all belong to the i/o system family. As a result an application program (or OS-module) can use these modules interchangeably. For instance, it can bypass the cache by using one of the low-level file systems directly. UNIX system calls concerned with i/o is mapped into standard EOS i/o-system entries by the system library procedures.

EOS-UNIX job control module:
This is the only special purpose module in the decentralized EOS-UNIX-kernel. It handles UNIX job and process management, i.e. login, fork, execute, signals, etc. It relies entirely on standard mechanisms in the EOS-kernel. UNIX system calls not concerned with i/o is mapped into entries in this module.

The Software Factory modules can be installed and removed on-the-fly. For instance, when demonstrating EOS, we usually show the following steps:

- Boot a system consisting of the EOS-kernel, the Object Directory, terminal and disc drivers, the VERSAdos file system, and a simple native EOS job handler (not part of the software factory).

- Using the native EOS job handler we execute a price-look-up application handling two terminals. We then install a cache file system on-the-fly and let one terminal use it.

The resulting performance improvement is striking. We also show how the new module responds properly when the application is aborted.

- After this we install the rest of the Software Factory modules and dynamically configure the EOS-UNIX job control module to use terminals not used by the native EOS job handler.

- The EOS-UNIX-kernel now runs concurrently with the native EOS job handler. In fact, applications running under either job handler can use much the same facilities (they can use the same decentral OS-modules). UNIX facilities like fork, exec, signals, etc. are, however, only available to applications running under the EOS-UNIX-kernel. Next, the native EOS job handler is removed to make all system resources available to the EOS-UNIX-kernel.

- After installation of a Window Terminal System (not part of the Software Factory), still without stopping the system, we show that it is possible to run UNIX from multiple windows on each terminal.

The first implementation of the EOS Software Factory has been made on a Motorola 68000 EXORmacs. It can relatively easy be moved to other hardware.

The EOS-kernel is written in assembly language and occupies about 30K bytes of storage. The OS-modules are written in EosPascal, an extended version of Pascal, and occupy about 150K bytes. The majority of UNIX applications and utilities are of course in "C", since we use the AT&T text unchanged. Each UNIX job occupies 1.5K bytes of system data, and each UNIX process 1K of system data.

The performance seems acceptable: On our first prototype the CPU-overhead to perform a disc file i/o is about 10 - 30 milliseconds (depending on the file systems used) on an 8 MHz MC68000. We expect these numbers to be halved when implementing planned improvements to the EOS-kernel.

Conclusion

We have introduced a new operating system technology that makes it possible to structure an operating system into decentral, protected modules. This technology has been used to implement an operating system, the EOS Software Factory, including a decentralized UNIX kernel.

While maintaining UNIX compatibility for existing tools and applications, the EOS Software Factory represents a significant improvement in flexibility.

Acknowledgements

The design and technology behind EOS was initiated at the University of Copenhagen by Eske B. Andersen, Ole Caprani, Søren Lauesen, Anders Peter Ravn, and Erik Reeh Nielsen.

The design presented in this paper was made at NCR by the authors and Peter Mikkelsen. The present implementation was made by the same four persons together with Charlotte Lunau and Edith Rosenberg.

References

[EOS 83]
S. Lauesen, E. Reeh Nielsen, V. Rosenqvist: EOS-Kernel Reference Manual.
NCR SE-Copenhagen Document No. 905-0000091.

[EOS 84]
EOS: System Programmers Introduction. NCR SE-Copenhagen Document No. 905-0000184.

[Wilkes 79]
M.V. Wilkes, R.M. Needham: The Cambridge CAP computer and its operating system. Elsevier North Holland, 1979.

[Wulf 74]
W. Wulf et.al.: Hydra: The kernel of a multiprocessor operating system. Comm. ACM 17,6 (June 1974) pp 337-345.

[Wulf 81]
W. Wulf et.al.: Hydra: C.mmp: An experimental computer system. McGraw-Hill, 1981.

Appendix: EOS kernel mechanims.

The most important EOS kernel mechanisms for module interaction will be illustrated by selected parts of an example OS-module that implements file reservation.

Object:

The interface of the file reservation module adheres to the standard i/o-system family architecture (cf. figure 1). Each open file is represented as an **object.** The file reservation module thus consists of a main i/o-system object and a variable number of file objects. Figure 3 shows the specification for the file objects. It includes a specification of the protected data of the objects (lowLevelFile and localData) and the code of the entry procedures (Read etc.).

The file reservation module uses another standard i/o-system for file storage. The file object used to store the contents of the file is referred to by the **kernel-pointer** (capability) lowLevelFile.

```
program ReservedFile object File
    with record ...
      lowLevelFile: ^^File;
      localData: ...
    end;
  entry Read {buffer, pos, count}
    ...
    begin
      ...
      lowLevelFile.Read
          (buffer, pos, count);
      ...
    end { Read } ;
  ...
end { ReservedFile } ;
```

Figure 3. File object implementation

Object Calls:

File i/o is done by calling the entry procedures of a file object (see figure 4). For UNIX programs these calls are part of the system call library procedures.

```
filel.Read (buf, readpos, count);
filel.Write(buf[1..count], writepos,
            ignorecount);
```

Figure 4. File object entry calls.

An entry procedure is called using the **object call** kernel operation. An object call appears much like an ordinary procedure call, but in fact it involves a change of hardware protection environment. When a process executes the code in figure 4 it cannot access the protected data of filel, it can only call the entry procedures. After calling filel.Read the process executes within the filel object. It can access the protected data and in turn call lowLevelFile to perform the actual i/o (cf. figure 3).

Object Creation:

To open a file the Assign entry in the i/o-system object of the file reservation module has to be called. The Assign entry creates a file object to represent the open file. Figure 5 shows the implementation of the Assign entry of the reservation module.

Multiple users may have read access to a file but to write to a file a user must have exclusive access. The "reserve" procedure lets the calling process wait until the required "iorights" can be granted (ignoring for this example the risk of deadlock).

```
entry Assign { file, name, iorights}
    ...
  begin
    ...
    Reserve (name, iorights);
    ...
    alloc.NewObj (file, ...);
      {Call allocator object to
       allocate space for the new
       file object}
    ...
    DeclGen (file, filedata, ...
             ReservedFile, ...
             Close, ... );
      {Declare new object to be a
       file object. "ReservedFile"
       specifies implementation, and
       "Close" is termination
       procedure}
    ...
    lowLevelFileSystem.Assign
      (filedata^^.lowLevelFile, ...)
      {Assign low-level file storing
       contents of file}
  end { Assign } ;
```

Figure 5. Assign entry.

The space for the new object is allocated by an allocator object. This may be the primitive allocator which is part of the EOS-kernel or it may be a more sophisticated module implemented on top of the primitive one.

The kernel operation **DeclGen** is used to declare the new object to be a file object. It defines the protected data and entry procedures of the object by specifying "ReservedFile" (cf. figure 3) to be the implementation of the object. It also specifies "Close" to be the termination procedure to be called when the object is deleted (see below).

As part of the initialization of the new file object the low-level file storing the contents is assigned.

Object Deletion:

An open file is closed by deleting the corresponding file object. This may be done explicitly (using the **Dealloc** kernel operation) or implicitly (as a result of the object assigning the file, e.g. a job, being deleted).

When an object is deleted the termination procedure specified during creation is automatically called. Figure 6 shows the termination procedure "Close" specified in figure 5.

```
private Close (file, filedata)
    ...
  begin
    ...
    Release (filedata^^.localdata);
      {Cancels reservation made by
       "reserve"}
    ...
  end { Close } ;
```

Figure 6. Termination procedure.

Once a file object has been created the EOS-kernel ensures that its termination procedure is called when it is deleted. Since a file object is deleted no later than the object creating it, i.e. the object opening the file, the reservation module gets the opportunity to clean up and cancel the reservation regardless of whether the user remembers to close the file. Note that "Close" does not explicitly delete "lowLevelFile", it is automatically deleted during deletion of the "ReservedFile" object.

Abortion:

When a module is **aborted** the entry procedures of the objects defined by the module is made empty, i.e. processes executing or calling the objects will return immediately.

As an example consider an eternal loop in the "Close" procedure of the reservation module. This will make a process trying to close (delete) a file loop until the reservation module is aborted. After the module has been aborted "Close" returns immediately. The deletion of the file object then automatically proceeds with the deletion of the low-level file object (including a call of the termination procedure in the low-level file system).

Processes that have called an aborted module does not necessarily return immediately, since they may have to perform some clean-up in non-aborted modules (as can be seen from the above example). However, the processes will return eventually, provided that these modules are ok.

Speed-Up:

An object called from an aborted object may not have to complete its operation in the ordinary way.

As an example consider the "Reserve" procedure of the file reservation module (figure 7). If the object opening a file is aborted there is no need to wait for the ioRights to become free, since the file will never be used.

When an object is aborted a **speed-up signal** is propagated into objects called by the aborted object. The speed-up signal can be tested explicitly or be allowed to continue to propagate so that later operations may be speeded-up, as in figure 7.

The "Reserve" procedure takes advantage of the fact that the "Wait" entry cancels its waiting and returns with a special result when receiving a speed-up signal. Note that even after being speeded-up the procedure has to do some clean-up, i.e. adjust "waitCount".

```
procedure Reserve ( name, ioRights )
    ...
  begin
    ...
    if rightsOccupied then begin
      waitCount := waitCount + 1;
      propagationMode (reject);
        {Allow speed-up signal to
         propagate so that "Wait"
         may be speeded-up}
      res := rightsFree.Wait;
        {May be speeded-up}
      propagationMode (stop);
        {Prevent speed-up signal
         from propagating}
      waitCount := waitCount - 1;
      if res = speededUp then
        Return; {Without creating
                 file object}
    end;
    ...
  end { Reserve } ;
```

Figure 7. Reserve procedure.